Maintaining Structural Invariants in Shape Analysis with Local Reasoning *

Sigmund Cherem and Radu Rugina

Computer Science Department
Cornell University
Ithaca, NY 14853
{siggi,rugina}@cs.cornell.edu

Abstract. This paper presents a novel shape analysis algorithm with local reasoning that is designed to analyze heap structures with structural invariants, such as doubly-linked lists. The algorithm abstracts and analyzes one single heap cell at a time. In order to maintain the structural invariants, the analysis uses a local heap abstraction that models the sub-heap consisting of one cell and its immediate neighbors. The proposed algorithm can successfully analyze standard doubly-linked list manipulations.

1 Introduction

Shape analyses are aimed at extracting heap invariants that describe the "shape" of recursive data structures [1]. For instance, heap reference count invariants allow a program analyzer to distinguish acyclic and unshared data structures, such as acyclic lists or trees, from structures with sharing or cycles. Shape information has many potential applications such as: verification of heap manipulations [2]; automatic parallelization [3]; static detection of memory leaks and other heap errors [4]; and compile-time memory management [5]. Statically computing reference count invariants is challenging because destructive heap mutations temporarily break these invariants. A shape analysis must determine that the invariants are restored as the destructive operations finish.

In recent work, we have developed a novel shape analysis framework that uses local reasoning about single heap cells [4]. In this framework, the analysis uses a local abstraction to describe the state of a single heap cell. Using the local abstraction, the analysis tracks the state of the single cell through the program, from the point where the cell is allocated, and up to the point where it becomes unreachable. The single cell is referred to as the tracked cell. As shown in [4], this approach makes it possible to build efficient intra-procedural and inter-procedural analysis algorithms. However, a shortcoming of the current formulation is that it cannot accurately compute shape information for an important class of heap structures – data structures with local structural invariants, such as doubly-linked lists or trees with parent pointers.

In this paper we present a shape analysis with local reasoning about single heap cells that is capable of identifying and maintaining information about structural invariants. We propose a new local abstraction capable of expressing such invariants. Then, we

* This work was supported in part by NSF grants CCF-0541217 and CNS-0406345.

develop an analysis algorithm that computes shape information using this abstraction. The local abstraction for a heap cell describes the local heap around the cell, consisting of the cell itself and its immediate neighboring cells. Points-to relations between the cell and its neighbors allow the analysis to express local structural invariants. The paper shows that maintaining structural invariants for the tracked cell requires knowledge about its neighbors' reference counts.

When a distant cell gets closer to the tracked cell and becomes one of its neighbors (for instance, when removing the element next to the tracked cell), a local analysis has no knowledge about the reference counts of the new neighbor. To address this issue, we propose an *assume-and-check* approach: when the analysis of a single cell reaches an assumption point in the program, it assumes facts about the neighbors' reference counts; at the same time, the analysis checks the reference counts of all tracked cells at that point, to ensure that the assumptions were correct.

The rest of the paper is organized as follows. Section 2 gives the background. Section 3 shows an example and discusses the issues that the analysis must overcome. Next, Section 4 presents the local abstraction and Section 5 shows the analysis algorithm. Finally, related work is discussed in Section 6 and we conclude in Section 7.

2 Background: Local Analysis of Single Heap Cells

This section discusses the key concepts behind heap analysis with local reasoning about single heap cells. The main idea is that the analysis uses a local abstraction to model one single heap cell at a time. Hence, the analysis has only local information about the one cell, but knows nothing about the rest of the heap. In contrast, traditional shape analyses that use shape graphs [6] or 3-valued logic [7] have a global view of the heap. Recent work has explored formulations using procedure-local sub-heaps [8], or using separation logic [9, 10]. Although these approaches restrict themselves to sub-heaps, their abstractions still describe entire structures (e.g., entire lists), not single cells.

Roughly speaking, an analysis that reasons about single cells is concerned with questions of the form "if property X holds for *one* heap cell before an operation, does X hold for that cell afterwards?". In contrast, global analyses answer questions of the form "if property X holds for *all* the cells before an operation, does X hold for all cells afterwards?". A local analysis is more efficient due to the finer granularity of the abstraction. However, it is more restricted because less information is available when analyzing a single cell.

The local abstraction of a heap cell is referred to as a *configuration*. The cell described by the configuration is referred to as *the tracked cell*. Each configuration contains reference counts for the tracked cell, plus additional information for accurately maintaining these reference counts. Reference counts are expressed relative to a *region partitioning* of the program's memory (both stack and heap) into a finite set of disjoint regions, so that each configuration keeps track of one reference count per region. To ensure a finite abstraction, reference counts are bound to a fixed value k per region (and a top value is used for larger counts). Usually, $k = 2$ suffices. In this paper, we assume a type-safe Java-like language, where a simple region partitioning can be constructed by using one region per variable and one region per heap field. In the rest of the paper,

```
class DLList {
   DLList n, p;
   int data;

   DLList(int d) {
     data = d;
   }
}
```

```
void insert(DLList x, int d) {
  DLList t;
  t = x.p;
  y = new DLList(d);
  y.n = x;
  y.p = t;
  t.n = y;
  x.p = y;
}
```

Fig. 1. Doubly-linked list insertion

we refer to regions using their variable or field names. The entire heap abstraction at a program point is the finite set of possible configurations at that point. However, configurations are independent, so they can be analyzed separately. The analysis uses efficient, fine-grained worklist algorithms to process individual configurations, not entire heap abstractions (in a fashion similar to attribute-independent analyses).

For a given program, the analysis generates a configuration after each allocation site, to model a *representative cell* created at that site. Then, the analysis tracks this configuration through the program using a dataflow analysis.

In our previous work [4, 5], each local abstraction is a triple (r, h, m), where r indicates the reference counts per region, h is a set of expressions that reference the tracked cell (or *hit* expressions); and m is a set of expressions that do not reference the cell (*miss* expressions). The h and m sets need not be complete; the richer these sets are, the more precise the analysis is. In general, redundant information is avoided, i.e., h and m exclude expressions e for which r already indicates whether e hits or misses.

For example, consider an acyclic singly-linked list, where next fields are named n. Assume that the first two list elements are pointed by variables x and y, respectively. This heap can be described using three local abstractions: $(x^1, \varnothing, \varnothing)$ describes the first list element; $(y^1 n^1, \{x.n\}, \varnothing)$ describes the second list element; and $(n^1, \varnothing, \{x.n\})$ describes one list element other than the first two, that is, it describes a representative among the cells in the tail of the list. Here, reference counts are described using superscripts, and missing regions have zero reference counts by default. The analysis can analyze each of these pieces separately, reasoning locally about each of them.

However, the triples (r, h, m) cannot express local structural invariants, such as doubly-linked list invariants. In this paper we propose a new local abstraction for describing and maintaining structural invariants.

3 Example

Consider the program in Figure 1. The program is written using a Java-like syntax and is used as a running example. The program inserts a new element y in a doubly-linked list, right before element x. Each list element has a field n that points to the next element, and a field p that points to the previous element. A correct manipulation of the list must maintain the doubly-linked list (DLL) invariant:

$$\forall h \,.\, (h.p \neq \mathsf{null} \Rightarrow h.p.n = h) \wedge (h.n \neq \mathsf{null} \Rightarrow h.n.p = h)$$

a) Before `insert(x)` b) After `insert(x)`

Fig. 2. Counterexample: the property rc for the shaded cell holds before `insert(x)`, but not after. The shaded box denotes the tracked cell. Solid lines are next links n, and dashed lines are previous links p.

3.1 Reference Counts and DLL Invariants

First, we show that maintaining precise heap reference counts requires knowledge about the DLL invariant. Consider the following two properties for a heap cell h in a doubly-linked list:

- $rc(h)$ is true if it has reference counts of at most 1 from each of the fields n and p;
- $dll(h)$ indicates that the DLL invariant holds for h.

We ask the following question: given a cell h such that $rc(h)$ holds, but $dll(h)$ might not hold before insert, does $rc(h)$ hold after the insertion? The answer is negative:

$$rc(h) \not\Rightarrow rc'(h)$$

where $rc(h)$ and $rc'(h)$ refer to the reference count property before and after insert. This is shown by the counterexample in Figure 2. A concrete heap before `insert(x)` is shown on the left of the figure, and the resulting heap after the insertion is shown on the right. The cell in question h (i.e., the tracked cell) is shown using the shaded box. Next links are shown using solid lines, and previous links are shown using dashed lines. The property $rc(h)$ holds before insert, but not after, because the cell pointed to by x has two references from n fields in the result heap.

Hence, the analysis must have knowledge about the DLL invariant in order to preserve accurate reference counts during destructive doubly-linked list operations. This is the case both for local and global analyses.

3.2 Maintaining the DLL Invariant Using Local Reasoning

Next we want to determine the amount of local information needed so that a local analysis can conclude that the DLL invariant is restored. We ask the following question: if one cell h is such that both $rc(h)$ and $dll(h)$ hold before insert, is it the case that $dll(h)$ also holds after insert? Note that nothing is known about the rc and dll properties of elements other than h. The answer to this question is again negative:

$$rc(h) \wedge dll(h) \not\Rightarrow dll'(h)$$

a) Before `insert(x)` b) After `insert(x)`

Fig. 3. Counterexample: a) before `insert(x)`, both the rc and dll properties hold for the shaded cell; b) after insertion, property dll doesn't hold for the shaded cell.

This is shown by the counterexample in Figure 3. The cell in question h is the shaded cell. In the heap before insert both $rc(h)$ and $dll(h)$ hold. However, the neighboring cell to the left of h is malformed because it is referenced by two p fields, one from the tracked cell and one from x. Inserting a new element before x "steals" a reference from h and breaks its DLL property: after insertion, $h.p.n \neq h$.

Still, it is possible to determine that insert maintains the DLL invariant using local reasoning. The required piece of information is that the neighbors $h.n$ and $h.p$ of the tracked cell h also satisfy the reference count property rc before insertion [1]. The analysis can then prove that if the tracked cell satisfies rc and dll before insert, and its neighbors satisfy rc, then rc and dll hold for the tracked cell after insert:

$$rc(h) \wedge dll(h) \wedge rc(h.p) \wedge rc(h.n) \;\Rightarrow\; rc'(h) \wedge dll'(h)$$

The goal of our analysis is to build an appropriate local abstraction and prove this property using that abstraction.

4 The Local Abstraction

Based on the above observations, the local heap abstraction must capture: a) local structural invariants, such as the dll property, and b) reference counts for both the tracked cell and its neighbors. We build such an abstraction as follows. The configuration for the tracked cell models the local heap consisting of that cell and its immediate neighbors, i.e., those heap cells that are pointed by, or point to the tracked cell. The configuration records the following information:

- Points-to relations between the tracked cell and its neighbors;
- Precise reference counts for the tracked cell, from each variable and each field; and
- Partial reference counts for the neighbors, from some variables and fields.

Graphically, a configuration can be thought as being a "circle" whose center is the tracked cell, and whose heap neighbors at distance 1 lay on this circle.

[1] A slightly weaker condition is actually sufficient: that $h.n$ has only one n reference, and $h.p$ has only one p reference, each of them from h.

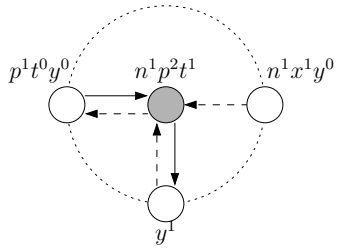

Fig. 4. Example local abstraction.

For instance, the local abstraction shown in Figure 4 arises during the analysis of `insert`. The tracked cell is the shaded node in the center. The points-to relations between the center node and its neighbors allow the analysis to express the local structural invariants. The reference counts from variables (x, y, or t) and fields (n or p) are shown using superscripts, for each node. The reference counts from variables can only be 0 or 1. For the tracked cell, the reference counts not shown (from x and y) are zero, by default. For the neighboring cells, the missing reference counts are unknown by default. Hence, reference counts are fully known for the tracked cell, but partially known for the neighbors. In the rest of the paper, we will refer to each local abstraction using the reference counts of the tracked cell. For instance, the above abstraction is $n^1p^2t^1$.

Note that the local abstraction does not contain summary nodes. In particular, nothing is known about the heap beyond the circle. This is the key aspect that distinguishes it from traditional global abstractions such as shape graphs.

4.1 Analysis of the Example

Figure 5 shows the analysis result for `insert` using this local abstraction. The possible local abstractions are shown at each point. In each abstraction, the tracked cell is shown as the shaded node. For simplicity, we consider only two input configurations at the entry of the function, n^1p^1 and $n^1p^1x^1$. The former describes a list cell that is not referenced by x; the latter is the cell that x references. Both cases assume that the cell in question is in the middle of the list. Four other configurations describe cases where the tracked cell is the first or the last element: n^1, n^1x^1, p^1, and p^1x^1. The analysis of those cases are similar and we omit them.

Consider the initial abstraction n^1p^1 and the first assignment `t = x.p`. The analysis tries to determine whether `x.p` is the tracked cell. Since there is not enough information to figure this out, the analysis bifurcates into two possible cases. These correspond to the first two columns in the figure. In the first case, `x.p` is not the tracked cell, so `t` will not reference the cell after the assignment. The resulting abstraction is n^1p^1. In the second case, `x.p` is the tracked cell, so t will reference it after the assignment. The resulting abstraction is $n^1p^1t^1$.

The analysis of `t = x.p` also infers that `x` does not reference the right neighbor in the first case (otherwise, `x.p` references the tracked cell); and that `x` references the

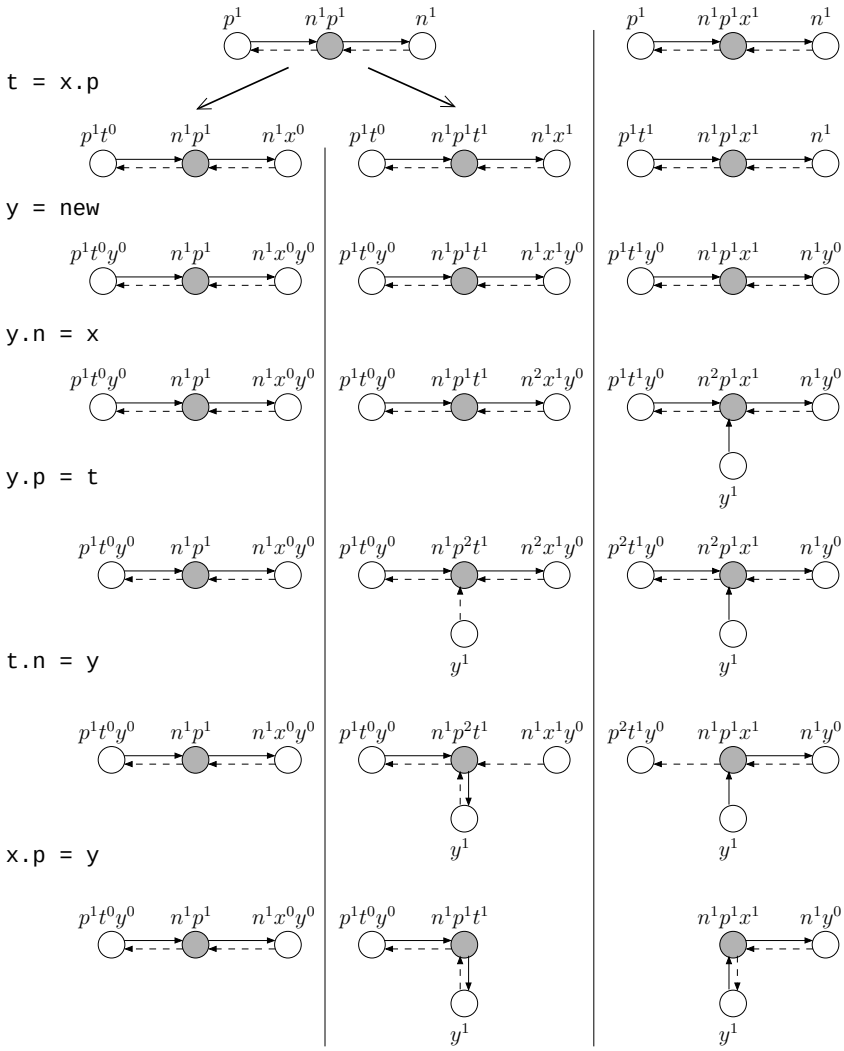

Fig. 5. Analysis of the example program.

right neighbor in the second case (because the only other cell that has a p field pointing to the tracked cell is the right neighbor). This information about x is needed later, when analyzing the assignment `x.p = y`.

Furthermore, in both cases the analysis of `t = x.p` infers that t doesn't reference the left neighbor, as shown by the reference count t^0. This is because the left neighbor has exactly one p reference, from the tracked cell. If t would point to the left neighbor, then x would reference the tracked cell, which is known to be false. Hence, the p^1 knowledge for the left neighbor allows the analysis to infer that t doesn't reference that neighbor. As a result, situations such as the one in Figure 3 are not possible.

The analysis of the other statements and local abstractions is similar. The configurations at the end of the function indicate that the rc and dll properties hold for all heap cells at that point.

Abstraction Model The local abstraction is modeled as a star graph S:

$$S = (V, v_o, v_{\mathsf{null}}, O, I, ||\cdot||) \qquad \text{where,}$$

$$v_o, v_{\mathsf{null}} \in V \qquad O \subseteq \mathsf{Field} \times V \qquad I \subseteq V \times \mathsf{Field} \qquad ||\cdot|| : V \to (\mathsf{Field} \cup \mathsf{Var}) \to \mathbb{N}_\top$$

The set V contains all nodes in the graph, where $v_o \in V$ is a distinguished center node representing the tracked cell. The node $v_{\mathsf{null}} \in V$ is a special node to represent null values. The set O contains outgoing edges from v_o. A pair $(f, v) \in O$ denotes the edge $v_o \to_f v$. The special edge $v_o \to_f v_{\mathsf{null}}$ indicates that the field f of the tracked cell is null. Similarly, an incoming edge $v \to_f v_o$ is denoted by a pair $(v, f) \in I$. The cardinality function $||\cdot||$ models the reference counts for each node in V. The set $\mathbb{N}_\top$ extends natural numbers with a special top value $\top$, such that $\top + 1 = \top - 1 = \top$. The heap reference count from a field f is denoted $||v||_f$. The reference counts from a variable x is denoted as $||v||_x$. If this value is not $\top$, it can only be 1 or 0, indicating whether the cell v is referenced by variable x or not. The special value $\top$ represents unknown information. As mentioned in Section 2, we use an upper bound k (e.g., $k = 2$) for the number of reference counts per field. In addition, the analysis uses a top configuration $S_\top$ to model cases where the analysis has lost precision about the tracked cell.

Given a configuration S, the analysis can derive the queries presented Figure 6:

- *Alias information*. Only nodes with consistent reference counts may be aliased.
- *Hit expressions*. The function *hit*(S, e, v) indicates that expression e references the cell represented by the node v. This is defined recursively using the reference counts and points-to relations.
- *Miss expressions*. The function *miss*(S, e, v) indicates that e doesn't reference the cell represented by v.

We will use these queries to formalize the analysis algorithm in the next Section. Figure 7 presents several invariants that our analysis maintains at all times:

1. All nodes other than v_{null} must be directly connected to v_o. Moreover, a node v pointing into v_o ($v \to_f v_o$) must also be pointed by v_o ($v_o \to_g v$) or by some variable ($||v||_x = 1$). This invariant ensures that the number of nodes and edges in the graph is bounded by the number of variables and fields in the program.

$$
\begin{aligned}
\mathit{mayAlias}(S, v_i, v_j) &\Leftrightarrow (v_i \neq v_o \neq v_j \ \wedge \\
&\qquad (\forall r \,.\, ||v_i||_r = ||v_j||_r \ \vee \ ||v_i||_r = \top \ \vee \ ||v_j||_r = \top)) \\
\mathit{hit}(S, e, v) &\Leftrightarrow \begin{cases} e = x \ \wedge ||v||_x = 1 \ \text{ or} \\ e = x.f \ \wedge ||v||_f \neq 0 \ \wedge \ (\exists v' \,.\, \mathit{hit}(S, x, v') \ \wedge \ v' \rightarrow_f v) \\ e = \mathsf{null} \wedge v = v_{\mathsf{null}} \end{cases} \\
\mathit{contains}(S, e) &\Leftrightarrow (\exists v \in V \,.\, \mathit{hit}(S, e, v)) \\
\mathit{miss}(S, e, v) &\Leftrightarrow \begin{cases} e = x \ \wedge ||v||_x = 0 \ \text{ or} \\ e = x.f \ \wedge ||v||_f = 0 \ \text{ or} \\ e = x \ \wedge (\exists v' \,.\, ||v'||_x = 1 \ \wedge \ \neg \mathit{mayAlias}(S, v, v')) \ \text{ or} \\ e = x.f \ \wedge ||v||_f = 1 \ \wedge \ (\exists v' \,.\, v' \rightarrow_f v \ \wedge \ \mathit{miss}(S, x, v')) \ \text{ or} \\ e = \mathsf{null} \wedge v \neq v_{\mathsf{null}} \ \text{ or} \\ e = x.f \ \wedge \mathit{hit}(S, x, v_o) \ \wedge \ v_o \rightarrow_f v_{\mathsf{null}} \ \wedge \ \neg \mathit{mayAlias}(S, v, v_{\mathsf{null}}) \end{cases}
\end{aligned}
$$

Fig. 6. Queries on configurations

$$
\begin{aligned}
&V = \{v_o, v_{\mathsf{null}}\} \cup \mathsf{range}(O) \cup \{v \mid v \in \mathsf{dom}(I) \ \wedge \ \exists x \,.\, ||v||_x = 1\} && (1) \\
&v_{\mathsf{null}} \notin \mathsf{dom}(I) && (2) \\
&\forall r \in \mathsf{Var} \cup \mathsf{Field} \,.\, ||v_o||_r \neq \top && (3) \\
&\forall v \in V, f \in \mathsf{Field} \,.\, |\{v \rightarrow_f v' \mid v' \in V\}| = 1 && (4) \\
&\forall v \,.\, ||v||_f = 1 \Rightarrow |\{v' \mid v' \rightarrow_f v\}| \leq 1 && (5) \\
&\forall v_1, v_2, e \,.\, \mathit{hit}(S, e, v_1) \wedge \mathit{hit}(S, e, v_2) \Rightarrow v_1 = v_2 && (6)
\end{aligned}
$$

Fig. 7. Consistency invariants maintained by the algorithm.

2. Since v_{null} represents null values, it can't have outgoing edges.
3. All references to the tracked cell are precisely known.
4. A node can have at most one outgoing edge with the same field.
5. If a node v has a single incoming reference from some field f, a configuration can only have one node to represent this predecessor.
6. Each expression references at most one node.

5 Analysis Algorithm

We now proceed to present the dataflow algorithm that computes a heap abstraction at each program point. For each configuration that models the state of the tracked cell before a statement, the analysis computes a set of configurations that describes the possible states of the cell after the statement.

We assume a simple program representation consisting of a control-flow graph whose nodes are simple assignment statements. Assignments and expressions have the

$$\textit{focusH}(S, x.f) = (V', v'_o, v'_{\mathsf{null}}, O', I'', ||\cdot||') \quad \text{where,}$$
$$S' = \begin{cases} \textit{unify}(\textit{addNode}(S, v_x, x, 1), v_x, v) & \neg\textit{contains}(S, x) \ \wedge\ ||v_o||_f = 1 \ \wedge\ v \rightarrow_f v_o \\ \textit{addNode}(S, v_x, x, 1) & \neg\textit{contains}(S, x) \quad v_x \text{ fresh} \\ \textit{unify}(S, v_x, v) & ||v_o||_f = 1 \ \wedge\ v \rightarrow_f v_o \\ S & \text{otherwise} \end{cases}$$
$$I'' = I' \cup \{v_x \rightarrow_f v_o\}$$

$$\textit{focusM}(S, x.f) = (V', v'_o, v'_{\mathsf{null}}, O', I'', ||\cdot||') \quad \text{where,}$$
$$S' = \begin{cases} \textit{unify}(\textit{addNode}(S, v', x, 0), v', v) & ||v_o||_f = 1 \ \wedge\ v \rightarrow_f v_o \quad v' \text{ fresh} \\ \textit{addNode}(S, v', x, 0) & ||v_o||_f = 1 \quad v' \text{ fresh} \\ S_\top & \text{otherwise} \end{cases}$$
$$I'' = I' \cup \{v' \rightarrow_f v_o\}$$

Fig. 8. Focusing operations. The helper functions *addNode* and *unify* are defined in Figure 11. We use S' as a shorthand notation for $(V', v'_o, v'_{\mathsf{null}}, O', I', ||\cdot||')$.

following form:

Statements	$s ::= x = \mathsf{new} \mid x = \mathsf{null} \mid x = y \mid x = y.f \mid x.f = y \mid x.f = \mathsf{null}$
Expressions	$e ::= \mathsf{null} \mid x \mid x.f$

where $x \in \mathsf{Var}$ ranges over variables, and $f \in \mathsf{Field}$ ranges over fields.

Initialization As discussed in Section 2, for each allocation site $x = \mathsf{new}$, the analysis builds a configuration $S = (\{v_o, v_{\mathsf{null}}\}, v_o, v_{\mathsf{null}}, \varnothing, \{v_o \rightarrow_f v_{\mathsf{null}} \mid f \in \mathsf{Field}\}, ||\cdot||)$ at the program point after the allocation, where $||v_o||_x = 1$ and $||v_o||_r = 0$ for any $r \neq x$. The configuration describes a representative heap cell allocated at this site. Then, the analysis tracks this configuration through the program.

Alternatively, if a code fragment is to be analyzed separately, the set of all possible configurations at the beginning of that fragment must be supplied.

Focus operations Given an input configuration describing the state of the tracked cell before an assignment statement $e_1 = e_2$, the analysis tries to determine whether e_1 and e_2 reference the tracked cell. Whenever the analysis cannot determine if e_i ($i \in \{1, 2\}$) hits or misses the tracked cell (i.e. $\neg\textit{hit}(S, e_i, v_o) \wedge \neg\textit{miss}(S, e_i, v_o)$), the analysis bifurcates and creates two new configurations that are focused with respect to e_i.

Figure 8 shows the focusing operations. Since exact reference counts are known for v_o, it is known whether variables hit or miss v_o. Therefore, the analysis only focuses expressions of the form $x.f$. To make an expression $x.f$ hit v_o, the analysis simply unifies the predecesor of v_o via field f (v) and the node referenced by x (v_x). The operation will also add the node v_x or the incoming field f if they didn't exist before focusing. A similar algorithm is used to make an expression $x.f$ miss v_o. Although, if $||v_o||_f \geq 2$, it is not possible to express the fact that $x.f$ misses the object. If this situation occurs, the focus operation returns an imprecise configuration $S_\top$ indicating that the analysis no longer tracks the state of the tracked cell.

$$
\begin{array}{lll}
\textit{transfer}(S, e_1 = e_2) = \textit{clean}(V', v'_o, v'_{\mathsf{null}}, O'', I'', ||\cdot||'') & & \text{where,} \\
S' & = \begin{cases} \textit{addNode}(\textit{kill}(S, e_1), v', y, 1) & e_1 = x.f \ \wedge \ e_2 = y \ \wedge \ \neg \textit{contains}(S, y) \\ \textit{kill}(S, e_1) & \text{otherwise} \end{cases} & \\
||v||''_r & = \begin{cases} ||v||'_r + 1 & \textit{hit}(S, e_2, v) \quad \wedge [(e_1 = x \wedge r = x) \ \vee \ (e_1 = x.f \wedge r = f)], \ \text{or} \\ ||v||'_r & \textit{miss}(S, e_2, v) \wedge (e_1 = x \wedge r \neq x) \quad \vee \ (r \neq f \wedge e_1 = x.f), \ \text{or} \\ \top & \text{otherwise} \end{cases} & \\
O'' & = O' \cup \{v_o \rightarrow_f v \mid e_1 = x.f \ \wedge \ \textit{hit}(S', x, v_o) \ \wedge \ \textit{hit}(S', e_2, v)\} & \\
I'' & = I' \ \cup \{v \rightarrow_f v_o \mid e_1 = x.f \ \wedge \ \textit{hit}(S', x, v) \ \wedge \ \textit{hit}(S', e_2, v_o)\} &
\end{array}
$$

Fig. 9. Transfer function. The helper functions *addNode*, *kill* and *clean* are defined in Figure 11. We use S' as a shorthand notation for $(V', v'_o, v'_{\mathsf{null}}, O', I', ||\cdot||')$.

$$
\begin{array}{lll}
\textit{merge}(S^1, S^2) = \textit{clean}(V', v'_o, v'_{\mathsf{null}}, O', I', ||\cdot||') & & \text{where,} \\
V' & = \{v_{i,j} \mid v_i \in V^1 \wedge v_j \in V^2\} & \\
v'_o & = v_{o,o} & \\
v'_{\mathsf{null}} & = v_{\mathsf{null},\mathsf{null}} & \\
O' & = \{v'_o \rightarrow_f v_{i,j} \mid v^1_o \rightarrow^1_f v_i \ \wedge \ v^2_o \rightarrow^2_f v_j\} & \\
I' & = \{v_{i,j} \rightarrow_f v'_o \mid v_i \rightarrow^1_f v^1_o \ \wedge \ v_j \rightarrow^2_f v^2_o\} & \\
||v_{i,j}||'_r & = ||v_i||^1_r \ \sqcup \ ||v_j||^2_r &
\end{array}
$$

Fig. 10. Merge operation. Precondition: $||v^1_o|| = ||v^2_o||$ and $(v^1_o \rightarrow^1_f v^1_o \Leftrightarrow v^2_o \rightarrow^2_f v^2_o)$.

Transfer function The analysis then applies the transfer function to each focused configuration. Figure 9 presents the transfer function for an assignment $e_1 = e_2$. First, the analysis nullifies e_1 using the helper function *kill*. For store assignments $x.f = y$, the analysis also creates the node for y in case it didn't exist, as this node might become a neighbor after the store. The reference counts are then updated. The appropriate reference count of each node v is increased when e_2 hits v, it remains unchanged when e_2 misses, and it is set to $\top$ when the analysis cannot determine whether e_2 hits or misses. The points-to edges are added in the case of store statements. Finally, the *clean* helper function removes nodes that are not neighbors of the tracked cell.

Merge operation At join points, the analysis uses the merge operation from Figure 10 to combine configurations from different branches. Two configurations are combined only if they have identical reference counts and the same set of self-edges on the tracked cell. The merge operation defines one node for each pair of nodes in the input configurations. The reference counts are combined using the join in the flat lattice $(\mathbb{N}_\top, \sqsubseteq)$. Thus, if $i \neq j$: $i \sqcup i = i$, $i \sqcup j = \top$, and $i \sqcup \top = \top \sqcup i = \top$. The *clean* operation guarantess that the number of nodes and edges in the resulting configuration is bounded by the number of variables and fields in the program.

Auxiliary functions The auxiliary operations used by the analysis are fairly straightforward. They are shown in Figure 11 and are summarized below:

$$
\begin{array}{lll}
\textit{addNode}(S, v', x, i) = & (V \cup \{v'\}, v_o, v_{\mathsf{null}}, O, I, ||\cdot||') \quad \text{where,} \\
& ||v||'_r = \begin{cases} i & r = x \ \wedge \ v = v' \\ \top & r \neq x \ \wedge \ v = v' \\ ||v||_r & \text{otherwise} \end{cases} \\
\textit{kill}(S, e) & = (V, v_o, v_{\mathsf{null}}, O - K, I - K, ||\cdot||') \text{ where,} \\
& ||v||'_r = \begin{cases} 0 & e = r = x \\ ||v||_r - 1 & e = x.f \ \wedge \ r = f \ \wedge \ \textit{hit}(S, e, v) \\ ||v||_r & \text{otherwise} \end{cases} \\
& K = \{v \rightarrow_f v' \mid e = x.f \ \wedge \ \neg \textit{miss}(S, x, v)\} \\
\textit{clean}(S) & = (V', v_o, v_{\mathsf{null}}, O, V' \lhd I, V' \lhd ||\cdot||) \text{ where,} \\
& V' = \{v_o, v_{\mathsf{null}}\} \cup \mathsf{range}(O) \cup \{v \mid v \in \mathsf{dom}(I) \ \wedge \ \exists x \,.\, ||v||_x = 1\} \\
& \text{where } V' \lhd f \text{ restricts the domain of } f \text{ to } V' \\
\textit{unify}(S, v_i, v_j) & = (V', v_o, v_{\mathsf{null}}, O', I', ||\cdot||') \text{ where,} \\
& V' = V - \{v_j\} \\
& O' = O - \{v_o \rightarrow_f v_j\} \cup \{v_o \rightarrow'_f v_i \mid v_o \rightarrow_f v_j \in O\} \\
& I' = I - \{v_j \rightarrow_f v_o\} \cup \{v_i \rightarrow'_f v_o \mid v_j \rightarrow_f v_o \in I\} \\
& ||v||'_r = \begin{cases} ||v_i||_r \sqcap ||v_j||_r & v = v_i \\ ||v||_r & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 11. Helper operations. The function *unify* assumes $\textit{mayAlias}(S, v_i, v_j)$ holds, and $v_j \neq v_{\mathsf{null}}$.

- The *addNode* operation adds a neighboring node, without connecting it to v_o. The reference count of the added node from variable x is set according to $i \in \{0, 1\}$. This function is used both when focusing and when applying the transfer function.
- The *kill* operation removes an expression, either x or $x.f$, from a configuration. Reference counts are updated accordingly. The operation supports strong updates when field expressions are killed.
- The *clean* operation removes unnecessary nodes from a configuration. This operation is used by the end of the transfer functions and merge operation.
- The *unify* operation combines two nodes that may alias into one single node. This is done by transferring all information from one node to the other. Moreover, the result has the most precise reference counts from the input nodes.

5.1 Assume-And-Check Approach

Although the analysis can successfully determine that the reference count property rc and the doubly-linked list invariant are preserved for the tracked cell during destructive operations, in many cases it cannot determine that the reference count property of the neighbors is restored. For instance, in the `insert` example from Figure 5 the heap reference counts are not known for the neighboring cell pointed by y, because y "came from the outside" to join the local heap. A similar situation occurs when removing an element from a list: a cell two levels of indirection away from the tracked cell gets

closer and becomes one of its neighbors. As discussed, the neighbor's reference count information is, however, needed before `insert`.

We address this issue using an assume-and-check approach. This approach is based on defining *assumption points* in the program. We consider that such points are manually marked by the user using a special `assume-and-check` instruction. The assumption points are program points where the analysis can safely restore the reference count information for the neighbors. As implied by the name, the analysis performs two tasks when it reaches such points:

- **Assume:** Whenever the analysis of a tracked cell reaches an assumption point, it assumes that the reference count property rc holds for all of its neighbors. More precisely, all neighbors are assumed to have at most one reference from each field. This enables the analysis to restore their reference counts: if the current configuration is such that the tracked cell points to neighbor v via some field f, i.e., $v_o \rightarrow_f v$, then the analysis restores v's reference count from f: $||v||_f = 1$.
- **Check:** Whenever the analysis of a tracked cell reaches an assumption point, it checks if the tracked cell itself satisfies the reference count property rc, i.e., if it has at most one reference per field. When the assumption is violated, the analysis reports an error and all of the analysis results are invalidated. Otherwise, if all checks succeed, then all assumptions were correct.

Essentially, restoring the reference counts of the neighbors requires knowledge about all cells. The assume-and-check approach provides a simple mechanism for gathering such global information without breaking the local analysis methodology.

Standard heap operations typically require one single assumption point, after the operation finishes. In the example from Section 3, an assume-and-check instruction is added at the end of the function. This suggests that default assumption at such points could be used to reduce the amount of annotations. In addition, assume-and-check instructions can be refined to indicate the specific field for which the reference count must be assumed and checked.

5.2 Soundness

We have formally proved that the analysis presented in this paper is sound with respect to the standard semantics of the language. The soundness result is summarized by the theorem below:

Theorem 1. *Given a program point, a concrete heap, and a set of configurations computed by the analysis at that point, each cell in the concrete heap is modeled by at least one of the configurations at that point.*

The reader is referred Appendix A for the concrete semantics of the language, the abstraction function, and the proof of the above soundness theorem.

5.3 Evaluation

We have developed a prototype implementation of the local analysis presented in this paper in Java, and used it to analyze the doubly-linked list programs shown in Table 1. Our local analysis has successfully verified that all of these programs maintain

Program	Local Abs.				Global Abs. (TVLA)		
	Configs. In	Avg.	Avg. Nodes per Config.	Time (sec)	Avg. Structures	Avg. Nodes per Struct.	Time (sec)
insertBefore	7	6.5	2.2	0.07	2.7	3.9	0.59
appendLast	4	4.5	2.1	0.06	4.6	3.7	0.77
concat	4	4.5	2.3	0.07	4.8	3.7	0.88
copy	4	4.5	2.1	0.09	4.8	3.5	1.24
insertNth	4	6.2	2.3	0.09	7.0	3.2	1.38
removeData	3	8.2	2.3	0.13	10.1	3.0	1.86
filter	3	26.3	2.0	0.37	24.7	2.2	4.19

Table 1. Analysis Evaluation

the doubly-linked list shape. All of the experiments were run on a 2GHz Pentium machine with 1GB of memory, running Linux.

The input to each program is described using at least 3 configurations (one for the middle, and one for each end of the list). Additional configurations are needed to indicate where the arguments point in the list. Programs that allocate new heap cells also include one configuration for the allocation site. The number of input configurations is shown in the first column of the table.

Each program, except `filter`, has been annotated with one single assume-and-check instruction, inserted at the end of the program. The `filter` program uses a loop to remove several elements from the input list. For this program, and additional assume annotation has been added at the beginning of the loop body. This ensures that the rc property holds on the neighboring cells after every removal from the list. The analysis successfully verifies the checks at all of the assumption points.

The data in Table 1 shows several statistics about our analysis: the average number of configurations per program point; the average number of nodes per configuration (excluding the null node); and the analysis running time. These results show that the analysis is fast, with an average running time of about 0.1 seconds per program.

To compare our implementation to a global analysis, we have also tested an implementation in TVLA [11]. We have added an instrumentation predicate to describe the DLL invariant. However, no global predicates, such as reachability, were included in this implementation. The right part of Table 1 shows the results obtained with TVLA. We observe that the number of 3-valued structures per program point is roughly equal to the number of configurations per program point in our analysis, but the number of nodes in those structures is larger than the number of nodes per configuration. Furthermore, the running time of the TVLA implementation is about 10 times slower. We attribute this in part to the fact that TVLA uses of a global abstraction, and in part to the fact that the TVLA engine is generic, while ours is specialized.

6 Related Work

The work on shape analysis dates back to Jones and Muchnick [12]. They developed a dataflow analysis for identifying (the lack of) cyclicity and sharing in heap structures us-

ing k-limited abstract heaps. Since then, many different approaches based on dataflow analysis and abstract interpretation have been proposed to address this problem [13–17, 6, 7, 18, 19, 8, 9]. Existing techniques include analyses that use path matrices and or matrices that describe reachability [14, 15, 17], reference counting analyses [13], analyses that use shape graphs [20, 18, 6], shape analyses and abstractions expressed using three-valued logic [21, 22, 7, 8]. In addition, heap verification techniques using model-checking or Hoare logic has also been explored [23–25]. Unlike abstract interpretation, logic-based tools rely on theorem provers and typically require heavyweight loop annotations. Alternatively, it is possible to synthesize loop invariants via predicate abstraction [26, 25]. The common aspect of all of the above techniques is that the analyzer or verifier requires a global view of the entire heap in order to analyze a particular piece of computation. In contrast, the analysis in this paper and our earlier analysis [4] are fundamentally different, as the analysis has knowledge about the local properties of one single heap cell, but is oblivious to the way the rest of the heap is structured. This fine-grained abstraction leads to efficient algorithms. This is achieved at the expense of giving up on global properties (such as reachability) that involve reasoning about unbounded sets of cells.

This paper follows our initial work on shape analysis with tracked heap cells [4]. The contribution of this work is a new local heap abstraction that expresses local structural invariants, and the development of an analysis that uses this abstraction to maintain these invariants. This algorithm makes shape analysis with local reasoning about single cells applicable to an important class of heap structures.

A related direction of research is the recent work on separation logic [27, 28]. This line of research has explored extensions of Hoare logic for reasoning about mutable heap structures, by providing features such as the separating conjunction and the frame rule, that makes it easier to write correctness proof for heap-manipulating programs. Recently, separation logic has also been applied to the shape analysis problem [10, 9]. Although the state transformers modify local portions of the abstract heap, their abstractions still describe entire linked structures. For instance, operations such as inserting or removing elements from a list require knowing that the entire list is well-formed, using a "listness" predicate ls. This predicate behaves similarly to the summary node in standard shape analyses; it describes a global invariant for the entire list, not a local property of a single cell.

7 Conclusions

We have presented an abstraction and analysis algorithm that makes it possible to apply shape analysis with local reasoning to data structures that maintain structural invariants, such as doubly-linked lists. The local abstraction of a cell describes the local heap around that cell, and is therefore able to express local structural invariants. The algorithm can successfully show that standard operations such as doubly-linked list insertions or removals maintain the doubly-linked list invariant.

References

1. Wilhelm, R., Sagiv, M., Reps, T.: Shape analysis. In: Proceedings of the 2000 International Conference on Compiler Construction, Berlin, Germany (2000)
2. Lev-ami, T., Reps, T., Sagiv, M., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: Proceedings of the 2000 International Symposium on Software Testing and Analysis. (2000)
3. Ghiya, R., Hendren, L., Zhu, Y.: Detecting parallelism in C programs with recursive data structures. In: Proceedings of the 1998 International Conference on Compiler Construction, Lisbon, Portugal (1998)
4. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: Proceedings of the 32th Annual ACM Symposium on the Principles of Programming Languages, Long Beach, CA (2005)
5. Cherem, S., Rugina, R.: Compile-time deallocation of individual objects. In: Proceedings of the International Symposium on Memory Management, Ottawa, Canada (2006)
6. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Transactions on Programming Languages and Systems **20**(1) (1998) 1–50
7. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002)
8. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Proceedings of the 12th International Static Analysis Symposium, London, UK (2005)
9. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vienna, Austria (2006)
10. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: The 13th International Static Analysis Symposium, Seoul, Korea (2006)
11. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Proceedings of the 7th International Static Analysis Symposium, Santa Barbara, CA (2000)
12. Jones, N., Muchnick, S.: Flow analysis and optimization of Lisp-like structures. In: Conference Record of the 6th Annual ACM Symposium on the Principles of Programming Languages, San Antonio, TX (1979)
13. Chase, D., Wegman, M., Zadek, F.: Analysis of pointers and structures. In: Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation, White Plains, NY (1990)
14. Hendren, L., Nicolau, A.: Parallelizing programs with recursive data structures. IEEE Transactions on Parallel and Distributed Systems **1**(1) (1990) 35–47
15. Hendren, L., Hummel, J., Nicolau, A.: A general data dependence test for dynamic, pointer-based data structures. In: Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation, Orlando, FL (1994)
16. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation, Orlando, FL (1994)
17. Ghiya, R., Hendren, L.: Is is a tree, a DAG or a cyclic graph? A shape analysis for heap-directed pointers in C. In: Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL (1996)
18. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Proceedings of the 10th International Static Analysis Symposium, San Diego, CA (2003)

19. Rugina, R.: Quantitative shape analysis. In: Proceedings of the 11th International Static Analysis Symposium, Verona, Italy (2004)
20. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL (1996)
21. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages, San Antonio, TX (1999)
22. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Proceedings of the 2001 International Conference on Compiler Construction, Genova, Italy (2001)
23. Moller, A., Schwartzbach, M.: The pointer assertion logic engine. In: Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation, Snowbird, UT (2001)
24. McPeak, S., Necula, G.: Data structure specification via local equality axioms. In: Proceedings of the 2005 Conference on Computer-Aided Verification, Seattle, WA (2005)
25. Lahiri, S., Qadeer, S.: Verifying properties of well-founded linked lists. In: Proceedings of the 33th Annual ACM Symposium on the Principles of Programming Languages, Charleston, SC (2006)
26. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation, Snowbird, UT (2001)
27. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark (2002)
28. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: Proceedings of the 28th Annual ACM Symposium on the Principles of Programming Languages, London, UK (2001)

A Soundness

This section presents the formalisms to conclude that our abstract semantics are sound. First, we present a concrete domain and semantics of our language. Then we present an abstraction function that relates concrete locations with their abstract counterparts. Then, we formalize the partial order that allows us to compare different abstract graphs. Finally, we formulate and prove our soundness theorem.

A.1 Concrete Domain and Semantics

We assume a simple domain based on a variable environment and a heap. All values are either a location or the null value.

$l \in \mathsf{Loc}$	Locations
$\varphi \in \mathsf{VEnv} = \mathsf{Var} \to \mathsf{Loc} \cup \{\mathsf{null}\}$	Variable environment
$h \in \mathsf{Heap} = \mathsf{Loc} \times \mathsf{Field} \to \mathsf{Loc} \cup \{\mathsf{null}\}$	Heap
$\sigma \in \mathsf{State} = \mathsf{VEnv} \times \mathsf{Heap}$	Concrete state

Given a location $l \in \mathsf{Loc}$, we write $l \in \sigma$ whenever $l \in \text{range}(\varphi) \cup \text{dom}(h) \cup \text{range}(h)$. We formalize the concrete semantics using the following denotational rules:

$$
\begin{aligned}
[\![x = y]\!](\varphi, h) &= (\varphi[x \mapsto \varphi(y)], h) \\
[\![x = y.f]\!](\varphi, h) &= (\varphi[x \mapsto h(\varphi(y), f)], h) \\
[\![x = \mathsf{null}]\!](\varphi, h) &= (\varphi[x \mapsto \mathsf{null}], h) \\
[\![x = \mathsf{new}]\!](\varphi, h) &= (\varphi[x \mapsto l^*], h[(l^*, f_1) \mapsto \mathsf{null}] ... [(l^*, f_n) \mapsto \mathsf{null}]) \\
[\![x.f = y]\!](\varphi, h) &= (\varphi, h[(\varphi(x), f) \mapsto \varphi(y)]) \\
[\![x.f = \mathsf{null}]\!](\varphi, h) &= (\varphi, h[(\varphi(x), f) \mapsto \mathsf{null}])
\end{aligned}
$$

where the allocation introduces a fresh location $l^* \notin \sigma$ and initializes all fields of such location ($\mathsf{Field} = \{f_1, ... f_n, \}$).

A.2 Abstraction Function

The abstraction of a location l under a concrete state σ is defined by:

$$\alpha_\sigma(l) = (V, v_o, v_{\mathsf{null}}, O, I, ||\cdot||)$$

where,

$$V = \{l, \mathsf{null}\} \cup \{l' \mid h(l, f) = l'\} \cup \{\varphi(x) \mid h(\varphi(x), f) = l\} \tag{1}$$

$$v_o = l \tag{2}$$

$$v_{\mathsf{null}} = \mathsf{null} \tag{3}$$

$$O = \{l \to_f h(l, f) \mid f \in \mathsf{Field}\} \tag{4}$$

$$I = \{l' \to_f l \mid h(l', f) = l \ \wedge \ f \in \mathsf{Field}\} \tag{5}$$

$$||v||_r = \begin{cases} 1 & r = x \ \wedge \ \varphi(x) = v \\ 0 & r = x \ \wedge \ \varphi(x) \neq v \\ i & r = f \ \wedge \ i = |\{l' \mid h(l', f) = v\}| \ \wedge \ i \leq k \\ \top & \text{otherwise} \end{cases} \tag{6}$$

The set of nodes in V consist of null, l and all neighbors of the location l that are either pointed from l or that are pointed by a variable. The location l is the center of the graph. Edges and reference counts are defined according to the state σ. The constant value k is used to bound the reference count values in the abstract domain.

The abstraction of an entire heap consists of the union of all the abstract locations:

$$\alpha(\sigma) = \{\alpha_\sigma(l) \mid l \in \mathsf{Loc} \cap (\text{range}(\varphi) \cup \text{dom}(h) \cup \text{range}(h))\}$$

To simplify the presentation of our proofs, the rest of this section will prove facts about one abstract location $\alpha_\sigma(l)$ at a time.

A.3 Abstraction equality

Two abstract locations S_1 and S_2 are said to be equivalent $S_1 \equiv S_2$ if there exists an isomorphism ($\approx$) between them, satisfying the following:

$$\begin{gathered} v_o^1 \approx v_o^2 \\ v_{\mathsf{null}}^1 \approx v_{\mathsf{null}}^2 \\ v_1 \approx v_2 \Rightarrow ||v_1||_r^1 = ||v_2||_r^2 \\ v_1 \approx v_2 \ \wedge \ v_1' \approx v_2' \Rightarrow (v_1 \rightarrow_f^1 v_1' \Leftrightarrow v_2 \rightarrow_f^2 v_2') \end{gathered}$$

In fact, our heap abstraction only needs to maintain a representative among equivalent abstract locations.

A.4 Partial Order

Let S_1 and S_2 be two star graphs. The graph S_1 is more precise than S_2, denoted $S_1 \sqsubseteq S_2$, if either $S_2 = S_\top$ or there exists some mapping $\mu : V_2 \rightarrow V_1$ that satisfies the following constraints:

$$\mu(v_o^2) = v_o^1 \tag{7}$$
$$\mu(v_{\mathsf{null}}^2) = v_{\mathsf{null}}^1 \tag{8}$$
$$||\mu(v_2)||_r \sqsubseteq ||v_2||_r \tag{9}$$
$$v_2 \rightarrow_f v_2' \Rightarrow \mu(v_2) \rightarrow_f \mu(v_2') \tag{10}$$

The center and null nodes must match (7, 8), and the most precise graph must include all reference counts (9) and points-to edges (10) that exist in the less precise graph. If such mapping μ exists, we call it a witness of the partial order relation.

Claim. $\sqsubseteq$ is in fact a partial order.

Proof. We need to show that is reflexive, transitive and antisymmetric. Reflexivity is trivially shown by choosing a mapping $\mu = \lambda v.v$. Transitivity is achieved by the composition of the witness mappings. This is, suppose $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_3$, let μ_a be the witness map of $S_1 \sqsubseteq S_2$, and μ_b the witness map of $S_2 \sqsubseteq S_3$, then $\mu_a \circ \mu_b$ is a witness map for $S_1 \sqsubseteq S_3$.

Finally, antisymmetry is shown with respect to our equivalence relation described in the previous section. Let μ_a be the witness of $S_1 \sqsubseteq S_2$ and let μ_b be the witness of $S_2 \sqsubseteq S_1$. We claim that $\mu_a \circ \mu_b$ is the identity function. This would mean that $\mu_a = \mu_b^{-1}$, hence the witness mappings represent an isomorphism between the abstract locations.

We now argue that $\mu_a \circ \mu_b$ is the identity, i.e $v = \mu_b(\mu_a(v))$. We need to consider the structural invariants imposed on our star graphs. We know that each abstract location S has 3 kinds of nodes: nodes explicitly named v_o and v_{null}, nodes pointed from the center, and nodes that are both pointing to the center and are hit by some variable. The first kind of nodes are mapped directly by definition of μ, and we directly know that $v_o = \mu_b(\mu_a(v_o))$. Consider now the second kind of node, this is a node v such that $v_o \rightarrow_f v$. By rule (10) of $S_1 \sqsubseteq S_2$ we know that $\mu_a(v_o) \rightarrow_f \mu_a(v)$, and using the same rule of $S_2 \sqsubseteq S_1$ we get that $\mu_b(\mu_a(v_o)) \rightarrow_f \mu_b(\mu_a(v))$. But $v_o = \mu_b(\mu_a(v_o))$ and v_o has a single out-edge with field f, hence $v = \mu_b(\mu_a(v))$. Finally, the third kind of nodes satisfy that $v \rightarrow_f v_o$ and also that there is some variable x, such that $||v||_x = 1$. If we apply the mappings and consider rule (9) we get that $||v||_x = ||\mu_b(\mu_a(v))||_x$. But, the invariant (6) of our abstraction (see Figure 7) is that only one node is hit by any expression, hence, only one node satisfies that $||v||_x = 1$. Thus, $v = \mu_b(\mu_a(v))$.

□

Claim. The function *merge* is in fact the join operator of this partial order.

Proof. We need to show that any S^1 or S^2 is more precise than *merge*(S^1, S^2). More over, we need to show that *merge*(S^1, S^2) is the most precise graph that satisfies this.

We can show that $S^1 \sqsubseteq$ *merge*(S^1, S^2) by defining a witness map μ as $\mu(v_{i,j}) = v_i^1$ for any $v_{i,j}$ in V'. The mapping is similar for S_2.

We need to show that for any other S such that $S^1 \sqsubseteq S$ and $S^2 \sqsubseteq S$, then *merge*$(S^1, S^2) \sqsubseteq S$. Let μ^1 and μ^2 be the witness mappings of the partial order relation $S^1 \sqsubseteq S$ and $S^2 \sqsubseteq S$, respectively. Then we define $\bar{\mu}(v) = v_{i,j}$ whenever $\mu^1(v) = v_i^1$ and $\mu^2(v) = v_j^2$. This mapping is a witness for *merge*$(S^1, S^2) \sqsubseteq S$. Hence we have shown that *merge*$(S^1, S^2) = S^1 \sqcup S^2$.

□

A.5 Soundness Theorem

Theorem 2. *Given a program point, let σ and A denote the concrete and abstract state computed for that program point by the concrete and abstract semantics, respectively. All location in σ is modeled by one or more abstract locations in A, this is*

$$\forall l \in \sigma \,.\, \exists S \in A \,.\, \alpha_\sigma(l) \sqsubseteq S \tag{11}$$

Proof. The theorem holds trivially for the entry point of the program, this is because at that point σ contains no locations. The rest of the proof will consist of showing two principles. First, we will show that once a location l^* is added to a state σ, an abstract location modeling l^* exists in its corresponding abstract state A. Lemma 1 proves this. Second, we will show a preservation principle, that is, a location correctly modeled before a statement must also be modeled correctly after the statement. We will show

this in two separate lemmas, showing that both the transfer functions (Lemma 2) and the focus operations (Lemma 3) satisfy this preservation property.

Lemma 1. *Consider an allocation statement $x = \mathsf{new}$. The freshly introduced location l^* is modeled by some abstract location S introduced by the abstract semantics, i.e.*

$$\alpha_{\sigma'}(l^*) \sqsubseteq S \tag{12}$$

Proof. The initialization of our algorithm introduces a configuration

$$S = (\{v_o, v_{\mathsf{null}}\}, v_o, v_{\mathsf{null}}, \varnothing, \{v_o \rightarrow_f v_{\mathsf{null}} \mid f \in \mathsf{Field}\}, ||\cdot||)$$

at the point after the allocation site, where $||v_o||_x = 1$ and $||v_o||_r = 0$ for any $r \neq x$, and $||v_{\mathsf{null}}||_r = \top$ for any r. We now show that it satisfies (12).

The proof is based on a mapping $\bar{\mu}$ defined as follows: $\bar{\mu}(v_o) = l^*$ and $\bar{\mu}(v_{\mathsf{null}}) = \mathsf{null}$. Clearly this map satisfies the conditions (7,8) of the partial order relation. The reference count condition (9) is trivially satisfied for v_{null} since $||v_{\mathsf{null}}||_r = \top$. But it is also satisfied for v_o since $||v_o||_r = ||l^*||_r$. Finally, the points-to edges condition (10) is also satisfied since the same edges appear in both $\alpha_{\sigma'}(l^*)$ and S graphs. In particular, both have an edge $v_o \rightarrow_f v_{\mathsf{null}}$ for every program field f.

□

Now we proceed to prove the preservation property for our transfer function.

Lemma 2. *Given a concrete state σ before a statement $e_1 = e_2$ and a concrete state σ' after the statement, i.e.*

$$[\![e_1 = e_2]\!]\sigma = \sigma' \tag{13}$$

Consider $l \in \sigma$ a heap location and $S \in A$ an abstract location, such that

$$\alpha_\sigma(l) \sqsubseteq S \tag{14}$$

Assuming that S is focused with respect to e_1 and e_2, then the star graph after the statement

$$S' = \mathit{transfer}(S, e_1 = e_2) \tag{15}$$

must satisfy that,

$$\alpha_{\sigma'}(l) \sqsubseteq S' \tag{16}$$

Proof. Whenever S or S' are $S_\top$, the lemma holds trivially. Hence we'll focus on the cases were this is not true.

We will prove this lemma by defining a witness mapping $\bar{\mu}$. The mapping $\bar{\mu}$ will allow us to compare $\alpha_{\sigma'}(l)$ and S'. We construct this mapping directly from the existing mapping μ used to compare $\alpha_\sigma(l)$ and S before the statement. The definition of $\bar{\mu}$ depends on the statement being analyzed:

$$\bar{\mu} = \begin{cases} V' \lhd \mu & e_1 = x \ \vee \ e_2 = \mathsf{null} \\ V' \lhd (\mu[n_y \rightarrow \varphi(y)]) & e_1 = x.f \ \wedge \ e_2 = y \ \wedge \ \neg \mathit{contains}(S, y) \end{cases} \tag{17}$$

where the node n_y is the node added by the transfer function to represent y. Since the transfer function uses the clean operation, some nodes may be removed from S, hence we update the domain of μ using the restriction operation $V' \lhd \mu$.

From this construction is immediate to see that $\bar{\mu}(v_o) = l$ and $\bar{\mu}(v_{\mathsf{null}}) = \mathsf{null}$. Hence we only need to check the other two conditions from the partial order definition. Hence to prove this lemma we just need to check the other two properties of witness mappings.

We will use four claims to prove them. First, we will show that query predicates used by the algorithm are also sound. Then, we will show the partial order, defined by $\bar{\mu}$, satisfies the reference count property (9). Finally, we will show that the partial order satisfies the points-to edges constraint (10).

Claim. Let S_1 and S_2 be two star graphs s.t. $S_1 \sqsubseteq S_2$, and let μ be a witness mapping of this relation. Let $v \in V_2$ and let v' and v'' be two nodes different than the center in V_2, then

$$\neg \textit{mayAlias}(S, v', v'') \Rightarrow \neg \textit{mayAlias}(S_1, \mu(v'), \mu(v'')) \tag{18}$$

$$\textit{hit}(S_2, e, v) \Rightarrow \textit{hit}(S_1, e, \mu(v)) \tag{19}$$

$$\textit{miss}(S_2, e, v) \Rightarrow \textit{miss}(S_1, e, \mu(v)) \tag{20}$$

Proof. Consider first the query *mayAlias* (18). Suppose that $\neg \textit{mayAlias}(S_2, v', v'')$ holds. Then, there exist some reference r such that both v' and v'' have a value different than $\top$ but also that $||v'||_r \neq ||v''||_r$. Since this value is not $\top$, by (9) we get that $||\mu(v')||_r = ||v'||_r$ and $||\mu(v'')||_r = ||v''||_r$. Hence $\neg \textit{mayAlias}(S_1, \mu(v'), \mu(v''))$ holds.

Now we proceed to show the implication for *hit* (19). We consider two cases, when $e = x$ and when $e = x.f$:

Case ($e = x$)		$\textit{hit}(S_2, x, v)$
From definition of *hit*	$\Rightarrow$	$\|\|v\|\|_x = 1$
By the partial order (9) and def. of μ	$\Rightarrow$	$\|\|\mu(v)\|\|_x = 1$
By definition of *hit*	$\Rightarrow$	$\textit{hit}(S_1, x, \mu(v))$
Case ($e = x.f$)		$\textit{hit}(S_2, x.f, v)$
By def. of *hit* we get for some v'	$\Rightarrow$	$v' \rightarrow_f v_o$
such that		$\|\|v'\|\|_x = 1$
By definition of μ we get	$\Rightarrow$	$\mu(v') \rightarrow_f \bar{\mu}(v)$
and also		$\|\|\mu(v')\|\|_x = 1$
By definition of *hit*	$\Rightarrow$	$\textit{hit}(S_1, x.f, \mu(v))$

We continue by proving the implication for *miss* (20). First, we study the case when $e = x$. The proof consists of 2 cases based on the two rules in the definition of *miss* for variables:

Case (explicit miss)		$\|\|v\|\|_x = 0$
By definition of μ	$\Rightarrow$	$\|\|\mu(v)\|\|_x = 0$
By definition of *miss*	$\Rightarrow$	$\textit{miss}(S_1, x, \mu(v))$

Case (implicit miss)		$\|\|v\|\|_x = \top$
and $\exists v'$ s.t. $\neg$*mayAlias*(S, v, v') and		$\|\|v'\|\|_x = 1$
Since $\|\|v_o\|\|_r \neq \top$ we get	$\Rightarrow$	$v \neq v_o$
But based on (18)	$\Rightarrow$	$\neg$*mayAlias*$(\alpha_\sigma(l), \mu(v), \bar{\mu}(v'))$
By def. of μ we also have		$\|\|\mu(v')\|\|_x = 1$
Together, by def of *miss* we have	$\Rightarrow$	*miss*$(S_2, x, \mu(v))$

When $e = x.f$ we need to consider 3 cases based on the definition of *miss*. There is one case for each rule where $x.f$ misses a node v:

Case (explicit miss)		$\|\|v\|\|_f = 0$
By partial order (9) and def. of μ	$\Rightarrow$	$\|\|\mu(v)\|\|_f = 0$
By definition of *miss*	$\Rightarrow$	*miss*$(S_2, x.f, \mu(v))$
Case (x misses unique predecessor)		$\|\|v\|\|_f = 1$
and there is a single v' s.t.		$v' \rightarrow_f v$
and also		$\|\|v'\|\|_x = 0$
By partial order (9) and def. of μ	$\Rightarrow$	$\|\|\mu(v)\|\|_f = 1$
and also		$\mu(v') \rightarrow_f \mu(v)$
and also		$\|\|\mu(v')\|\|_x = 0$
By definition of *miss*	$\Rightarrow$	*miss*$(S_2, x.f, \mu(v))$
Case ($x.f$ is null)		$\neg$*mayAlias*$(S_1, v, v_{\mathsf{null}})$
and there is a v' s.t.		$\|\|v'\|\|_x = 1$
and also		$v' \rightarrow_f v^1_{\mathsf{null}}$
By partial order and def. of μ we get	$\Rightarrow$	$\neg$*mayAlias*$(S_2, \mu(v), v^2_{\mathsf{null}})$
and also		$\|\|\mu(v')\|\|_x = 1$
and also		$\mu(v') \rightarrow_f v^2_{\mathsf{null}}$
By definition of *miss* we get	$\Rightarrow$	*miss*$(S_2, x.f, \mu(v))$

In all cases we shown that *miss* is preserved by the partial order.

□

Claim. For some variable x, field f and node v in the abstract graph S. Given that $\alpha_\sigma(l) \sqsubseteq S$, the following always hold,

$$\textit{hit}(S, x, v) \Rightarrow \mu(v) = \varphi(x) \tag{21}$$

$$\textit{miss}(S, x, v) \Rightarrow \mu(v) \neq \varphi(x) \tag{22}$$

$$\textit{hit}(S, x.f, v) \Rightarrow \mu(v) = h(\varphi(x), f) \tag{23}$$

$$\textit{miss}(S, x.f, v) \Rightarrow \mu(v) \neq h(\varphi(x), f) \tag{24}$$

Proof. First we prove (21):

Assume that		*hit*(S, x, v)
By previous claim (19) we know	$\Rightarrow$	*hit*$(\alpha_\sigma(l), x, \mu(v))$
By definition of *hit*	$\Rightarrow$	$\|\|\mu(v)\|\|_x = 1$
By abstraction function (6) we get	$\Rightarrow$	$\mu(v) = \varphi(x)$

The proof for (22) requires us to consider 2 cases:

Assume that		$\mathit{miss}(S, x, v)$
By previous claim (20) we know	$\Rightarrow$	$\mathit{miss}(\alpha_\sigma(l), x, \mu(v))$
Case (explicit miss)		$\|\|\mu(v)\|\|_x = 0$
By abstraction function (6) we get	$\Rightarrow$	$\mu(v) \neq \varphi(x)$
Case (implicit miss)	$\Rightarrow$	$\neg\mathit{mayAlias}(\alpha_\sigma(l), \mu(v), v')$
for some v' such that		$\|\|v'\|\|_x = 1$
By abstraction function (6)	$\Rightarrow$	$\neg\mathit{mayAlias}(\alpha_\sigma(l), \mu(v), \varphi(x))$
By def. of *mayAlias*	$\Rightarrow$	$\mu(v) \neq \varphi(x)$
In all cases we showed	$\Rightarrow$	$\mu(v) \neq \varphi(x)$

We prove now the case when $x.f$ hits (23):

Assume that		$\mathit{hit}(S, x.f, v)$
By previous claim (19)	$\Rightarrow$	$\mathit{hit}(\alpha_\sigma(l), x.f, \mu(v))$
By definition of *hit* we get	$\Rightarrow$	$\|\|\mu(v')\|\|_x = 1$
and also		$\mu(v') \rightarrow_f \bar{\mu}(v)$
By abstraction function (4)	$\Rightarrow$	$\mu(v) = h(\varphi(x), f)$

Finally, we proof the case when $x.f$ misses (24), we need to consider 3 cases:

Assume that		$\mathit{miss}(S, x.f, v)$
By previous claim (20)	$\Rightarrow$	$\mathit{miss}(\alpha_\sigma(l), x.f, \mu(v))$
Case (explicit miss)		$\|\|\mu(v)\|\|_f = 0$
By abstraction (6)	$\Rightarrow$	$\forall l' \,.\, h(l', f) \neq \mu(v)$
In particular for $l' = \varphi(x)$	$\Rightarrow$	$\mu(v) \neq h(\varphi'(x), f)$
Case (x misses unique predecessor)		$\|\|\mu(v)\|\|_f = 1$
and also		$\mu(v') \rightarrow_f \mu(v)$
and also		$\|\|\mu(v')\|\|_x = 0$
By abstraction function (6)	$\Rightarrow$	$\mu(v') \neq \varphi(x)$
and also		$h(\mu(v'), f) = \bar{\mu}(v)$
and also		$\|\{l' \mid h(l', f) = \mu(v)\}\| = 1$
Which combined means that	$\Rightarrow$	$\mu(v) \neq h(\varphi'(x), f)$
Case ($x.f$ is null)		$\neg\mathit{mayAlias}(S_2, \mu(v), v^2_{\mathsf{null}})$ (25)
and also		$\|\|\mu(v')\|\|_x = 1$
and also		$\mu(v') \rightarrow_f v^2_{\mathsf{null}}$
By abstraction function (6)	$\Rightarrow$	$\mu(v') = \varphi(x)$
and also		$h(\mu(v'), f) = \mathsf{null}$
By (25) we know $\mu(v) \neq \mathsf{null}$, hence	$\Rightarrow$	$\mu(v) \neq h(\varphi'(x), f)$
In all cases we showed that	$\Rightarrow$	$\mu(v) \neq h(\varphi'(x), f)$

This concludes the proof of the claim.

□

Claim. After the assignment the reference counts are consistent, as described by the Equation 9, i.e. $||\bar{\mu}(v)||'_r \sqsubseteq ||v||'_r$.

Proof. We show the property holds for each kind of assignment.

– Assignments of the form $x = e_2$. The transfer function will change $||v||'_x$ to be either 1 if e_2 hits the node v, 0 if e_2 misses this node, and $\top$ if it can't tell if e_2 hits or misses. We only need to consider the cases where $||v||'_x \neq \top$, because when $||v||'_x = \top$ then the property $||\bar{\mu}(v)||'_x \sqsubseteq ||v||'_x$ holds trivially. We now consider the other two cases:

Case (e_2 hits)		$\mathit{hit}(S, e_2, v)$	
By abstract semantics	$\Rightarrow$	$\|\|v\|\|'_x = 1$	(26)
Consider a case for each rule in hit			
Sub-case ($e_2 = y$)		$e_2 = y$	
By previous claim (21) and def. of $\bar{\mu}$	$\Rightarrow$	$\bar{\mu}(v) = \varphi(y)$	
By concrete semantics (13)	$\Rightarrow$	$\bar{\mu}(v) = \varphi'(x)$	
Sub-case ($e_2 = \mathsf{null}$)		$e_2 = \mathsf{null}$	
By definition of $\bar{\mu}$ and abstract function (3)	$\Rightarrow$	$\bar{\mu}(v) = \mathsf{null}$	
By concrete semantics (13)	$\Rightarrow$	$\bar{\mu}(v) = \varphi'(x)$	
Sub-case ($e_2 = y.f$)		$e_2 = y.f$	
By previous claim (23)	$\Rightarrow$	$\bar{\mu}(v) = h(\varphi(y), f)$	
By concrete sem. (13)	$\Rightarrow$	$\bar{\mu}(v) = \varphi'(x)$	
From all sub-cases	$\Rightarrow$	$\bar{\mu}(v) = \varphi'(x)$	
By abstraction function (6)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_x = 1$	
Combined with (26)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|_x \sqsubseteq \|\|v\|\|'_x$	
Case (e_2 misses)		$\mathit{miss}(S, e_2, v)$	
By abstract semantics	$\Rightarrow$	$\|\|v\|\|'_x = 0$	(27)
Consider a case for each rule in *miss*			
Sub-case ($e_2 = y$)		$e_2 = y$	
By previous claim (22)	$\Rightarrow$	$\bar{\mu}(v) \neq \varphi(y)$	
By concrete semantics (13)	$\Rightarrow$	$\bar{\mu}(v) \neq \varphi'(x)$	
Sub-case ($e_2 = \mathsf{null}$)		$e_2 = \mathsf{null}$	
and		$v \neq v_{\mathsf{null}}$	
By definition of $\bar{\mu}$		$\bar{\mu}(v) \neq \mathsf{null}$	
By concrete semantics (13)	$\Rightarrow$	$\varphi'(x) = \mathsf{null}$	
Hence	$\Rightarrow$	$\bar{\mu}(v) \neq \varphi'(x)$	

Sub-case ($e_2 = y.f$)		$e_2 = y.f$
By previous claim (24) we get	$\Rightarrow$	$h(\varphi(y), f) \neq \bar{\mu}(v)$
By concrete semantics (13)	$\Rightarrow$	$\bar{\mu}(v) \neq \varphi'(x)$

From all sub-cases we get		$\bar{\mu}(v) \neq \varphi'(x)$
By abstraction function (6)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_x = 0$
Combined with (27)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|_x \sqsubseteq \|\|v\|\|'_x$

– Assignments $x.f = y$. The transfer function will update the reference count from field f. As before we ignore the trivial case when $||v||'_f = \top$. This automatically handles the case when the node of y is not present in the star graph S. It also handles any case where $||v||_f = \top$ before the statement.

The reference count of a node v will increase by one $||v||'_f = ||v||_f + 1$ if y hits and $x.f$ does misses the node v in S. It will decrease by one if $x.f$ hits and y misses. And it will stay the same in two situations, when y and $x.f$ both hit the node, and $x.f$ and y miss. We need to consider what happens on each of the four situations.

Case (adding a reference)		$\mathit{hit}(S, y, v) \ \wedge \ \mathit{miss}(S, x.f, v)$	
By abstract semantics	$\Rightarrow$	$\|\|v\|\|'_f = \|\|v\|\|_f + 1$	(28)
Since y hits, claim (21) shows		$\bar{\mu}(v) = \varphi(y)$	(29)
Since $x.f$ misses, claim (24) shows		$\bar{\mu}(v) \neq h(\varphi(x), f)$	
Let $P = \{l' \mid h(\varphi(x), f) = \bar{\mu}(v)\}$	$\Rightarrow$	$\varphi(x) \notin P$	(30)
By abstract function (6)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|_f = \|P\|$	
By concrete sem. (13) and (29)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f = \|P \cup \{\varphi(x)\}\|$	
Since (30) shows $\varphi(x) \notin P$	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f = \|P\| + 1$	
By (14), $\|\|\bar{\mu}(v)\|\|_f \sqsubseteq \|\|v\|\|_f$ with (28)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f \sqsubseteq \|\|v\|\|'_f$	

Case (removing a reference)		$\mathit{miss}(S, y, v) \ \wedge \ \mathit{hit}(S, x.f, v)$	
By abstract semantics	$\Rightarrow$	$\|\|v\|\|'_f = \|\|v\|\|_f - 1$	(31)
Since y misses, claim (22) shows		$\bar{\mu}(v) \neq \varphi(y)$	(32)
Since $x.f$ hits, claim (23) shows		$\bar{\mu}(v) = h(\varphi(x), f)$	
Let $P = \{l' \mid h(\varphi(x), f) = \bar{\mu}(v)\}$	$\Rightarrow$	$\varphi(x) \in P$	(33)
By abstract function (6)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|_f = \|P\|$	
By concrete sem. (13) and (32)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f = \|P - \{\varphi(x)\}\|$	
Since (33) shows $\varphi(x) \in P$	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f = \|P\| - 1$	
By (14), $\|\|\bar{\mu}(v)\|\|_f \sqsubseteq \|\|v\|\|_f$ with (31)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f \sqsubseteq \|\|v\|\|'_f$	

Case (maintaining the reference)		$\mathit{hit}(S, y, v) \wedge \mathit{hit}(S, x.f, v)$	
By abstract semantics	$\Rightarrow$	$\|\|v\|\|'_f = \|\|v\|\|_f$	(34)
Since y hits, claim (21) shows		$\bar{\mu}(v) = \varphi(y)$	(35)
Since $x.f$ hits, claim (23) shows		$\bar{\mu}(v) = h(\varphi(x), f)$	
Let $P = \{l' \mid h(\varphi(x), f) = \bar{\mu}(v)\}$,			
by abstract function (6)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|_f = \|P\|$	
By concrete sem. (13) and (35)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f = \|P\|$	
By (14), $\|\|\bar{\mu}(v)\|\|_f \sqsubseteq \|\|v\|\|_f$ with (34)	$\Rightarrow$	$\|\|\bar{\mu}(v)\|\|'_f \sqsubseteq \|\|v\|\|'_f$	

The fourth case is very similar to this last case, so we omit it. As in this case, the set P of predecessors of $\bar{\mu}(v)$ via field f is not changed by the assignment. The main difference with this case is that $\varphi(x) \notin P$ neither before nor after the statement.

– The proof for assignments $x.f = \mathsf{null}$ is similar to $x.f = y$.

We have shown that the reference counts is maintained consistent through all possible assignments.

□

Claim. After the assignment the points-to edges are also consistent: $v \rightarrow'_f v' \Rightarrow \bar{\mu}(v) \rightarrow'_f \bar{\mu}(v')$.

We consider all possible assignments in the program. Only store assignments can change the set of edges I and O, hence all other assignments trivially maintain the points-to edge consistency property. Hence, we consider assignments $x.f = y$, when either x or y hits the center (so that an edge is reflected either in I or O).

Case (x hits the center)		$\mathit{hit}(S, x, v_o)$	(36)
and		$\mathit{hit}(S, y, v)$	(37)
By abstract semantics, an out edge is added	$\Rightarrow$	$v_o \rightarrow'_f v$	
By (36) and claim (21)	$\Rightarrow$	$\bar{\mu}(v_o) = \varphi(x)$	
By (37) and claim (21)	$\Rightarrow$	$\bar{\mu}(v) = \varphi(y)$	
By concrete semantics (13)	$\Rightarrow$	$h'(\bar{\mu}(v_o), f) = \bar{\mu}(v)$	
By abstraction function (4)	$\Rightarrow$	$\bar{\mu}(v_o) \rightarrow'_f \bar{\mu}(v)$	

Hence we have showed that an out edge $v_o \rightarrow'_f v$ added by the abstract semantics, is consistently added by the concrete semantics $\bar{\mu}(v_o) \rightarrow'_f \bar{\mu}(v)$. The case for in-edges is similar and we omit it. Also, assignments of the form $x.f = \mathsf{null}$ have a similar proof.

□

Finally, from these last two claims we have shown that (16) holds. Hence we have shown that the transfer functions are sound.

□

Now we proceed with our last lemma, to show that our focus operations are sound.

Lemma 3. *Let l be a concrete location, and S an abstract location such that*

$$\alpha_\sigma(l) \sqsubseteq S \tag{38}$$

Consider an expression $x.f$, such that hit$(S, x.f)$ = miss$(S, x.f)$ = false*. And let $S_h =$ focusH$(S, x.f)$ and $S_m =$ focusM$(S, x.f)$. Then, the following holds:*

$$h(\varphi(x), f) = l \quad \Rightarrow \quad \alpha_\sigma(l) \sqsubseteq S_h \tag{39}$$

$$h(\varphi(x), f) \neq l \quad \Rightarrow \quad \alpha_\sigma(l) \sqsubseteq S_m \tag{40}$$

Proof. Consider first the case when $h(\varphi(x), f) = l$. If no unification occurs during focusing, the result is trivially consistent. Let v_x be a node in S_h such that $||v_x||_x = 1$. Consider the case when v_x was already present before focusing, and v and v_x are unified. We want to show that the reference counts and edges involving these nodes are still consistent. We first show that the witness mapping μ satisfies that $\mu(v) = \mu(v_x)$. This is true because unify is only called when $||v_o||_f = 1$ and $v \rightarrow_f v_o$:

$$\begin{array}{lcll} & & ||v_o||_f = 1 & \\ \text{By definition of } \mu \text{ and } \mu(v_o) = l & \Rightarrow & ||l||_f = 1 & \\ \text{Since } h(\varphi(x), f) = l & \Rightarrow & \{l' \mid h(l', f) = l\} = \{\varphi(x)\} & (41) \\ & & v \rightarrow_f v_o & \\ \text{By definition of } \mu & \Rightarrow & \mu(v) \rightarrow_f v_o & \\ \text{By abstract function (5)} & \Rightarrow & \mu(v) \in \{l' \mid h(l', f) = l\} & \\ \text{By (41) the only option is} & \Rightarrow & \mu(v) = \varphi(x) & \\ \text{By definition of } \mu \text{ and abs function} & \Rightarrow & \mu(v) = \mu(v_x) & \end{array}$$

Since $\mu(v) = \mu(v_x)$ it follows that the reference counts and edges remain consistent. Consider for example some reference r, such that $||v||'_r = i \neq \top$ after the unification. Then, before the unification $||v||_r = i$ or $||v_x||_r = i$, or both. Hence, one of v_x or v can be used to show that $||\mu(v)||_r = i$. Similarly, an edge $v \rightarrow'_g v_o$ after focusing could be based on either an existing edge $v \rightarrow_g v_o$, an existing edge $v_x \rightarrow_g v_o$, or the new edge added during focusing ($v_x \rightarrow_f v_o$). Since both v and v_x are mapped to the same node, we know that $\mu(v) \rightarrow_g v'$ holds in the first two cases, and hence the edge $v \rightarrow'_g v_o$ is consistently represented in the concrete state. The third case is trivially handled since the added edge matches our assumption that $h(\varphi(x), f) = l$. A similar argument applies for out edges $v_o \rightarrow'_g v$.

When v_x was not present in the star graph, we add it. We define a mapping $\bar{\mu}$ by adding this node to μ, $\bar{\mu} = V_h \lhd \mu[v_x \mapsto \varphi(x)]$. Then the proof follows the same structure as before.

We have shown that (39) holds, we now consider the case when $h(\varphi(x), f) \neq l$. The argument in this case is quite similar to the previous case. In this case the node v is updated to have $||v||_x = 0$. This is consistent with the concrete state because $\varphi(x) \notin \{l' \mid h(l', f) = l\} = \{\mu(v)\}$, hence $\mu(v) \neq \varphi(x)$.

The special case when the edge $v' \rightarrow_f v_o$ is added is when $||v_o||_f = 1$ but no edge $v \rightarrow_f v_o$ exists. In this case we extend the mapping μ to $\bar{\mu} = \mu[v' \rightarrow l']$ such that $h(l', f) = l$. This mapping satisfies that $\bar{\mu}(v') \neq \varphi(x)$. It becomes immediate that the partial order relation $\alpha_\sigma(l) \sqsubseteq S_m$ holds using $\bar{\mu}$. Therefore, (40) holds.

In conclusion we have shown that the focusing operations are sound.

□

Finally, this concludes the proof of our soundness theorem □